

Sequential Data Structures, Part 1

Lists in Racket

Please take a handout &
Sit in row H or forward

But first, let's review recursion

Suppose we want a Racket function `halve-count`.

Input: a strictly positive integer n .

Output: The number of times we must divide n by 2
until we reach 1. (i.e., an integer approximation to $\log_2 n$.)

$$\text{halve-count}(n) = \begin{cases} 0 & \text{base case} \\ 1 + \text{halve-count}(\lfloor \frac{n}{2} \rfloor) & \text{otherwise} \end{cases}$$

recursive step

But first, let's review analysis

Let's analyze this function, mathematically!

1. What's our *cost metric*, i.e., what are we counting?
2. What mathematical tool should we use?

```
(define (halve-count n)
```

```
  (if (= n 1)
```

```
      0
```

```
      (+ 1 (halve-count (quotient n 2)))))
```

base case

recursive step

$$T(1) = 0 \text{ adds}$$

$$T(n) = 1 + T(n/2) \text{ adds}$$

Ask for help solving, if needed

In CS 60, we only need to know how to set up recurrence relations, not solve them.

The screenshot shows the WolframAlpha website interface. At the top, the browser's address bar displays the URL: `www.wolframalpha.com/input/?i=T(1)+%3D0,+T(n)+%3D1+%2B+T(n%2F2)`. The WolframAlpha logo is centered, with the tagline "computational knowledge engine". Below the logo, the input field contains the recurrence relation: $T(1) = 0, T(n) = 1 + T(n/2)$. To the right of the input field are links for "Examples" and "Random". Below the input field, the "Input:" section shows the recurrence relation: $T(1) = 0 \mid T(n) = T\left(\frac{n}{2}\right) + 1$. The "Recurrence equation solution:" section displays the solution: $T(n) = \frac{\log(n)}{\log(2)}$. A handwritten orange arrow points from the text $\log_2 n$ to the solution. Below the solution, a note states: "log(x) is the natural logarithm". At the bottom left, there is a "Download page" link. At the bottom right, it says "POWERED BY THE WOLFRAM LANGUAGE". On the right side of the page, there is a sidebar with the text "Take Wolfram|Alpha anywhere." and icons for mobile devices, a "PRO" badge, and a settings icon.

WolframAlpha computational knowledge engine

$T(1) = 0, T(n) = 1 + T(n/2)$

Examples Random

Input:

$$T(1) = 0 \mid T(n) = T\left(\frac{n}{2}\right) + 1$$

Recurrence equation solution:

$$T(n) = \frac{\log(n)}{\log(2)} \leftarrow \log_2 n$$

log(x) is the natural logarithm

Download page

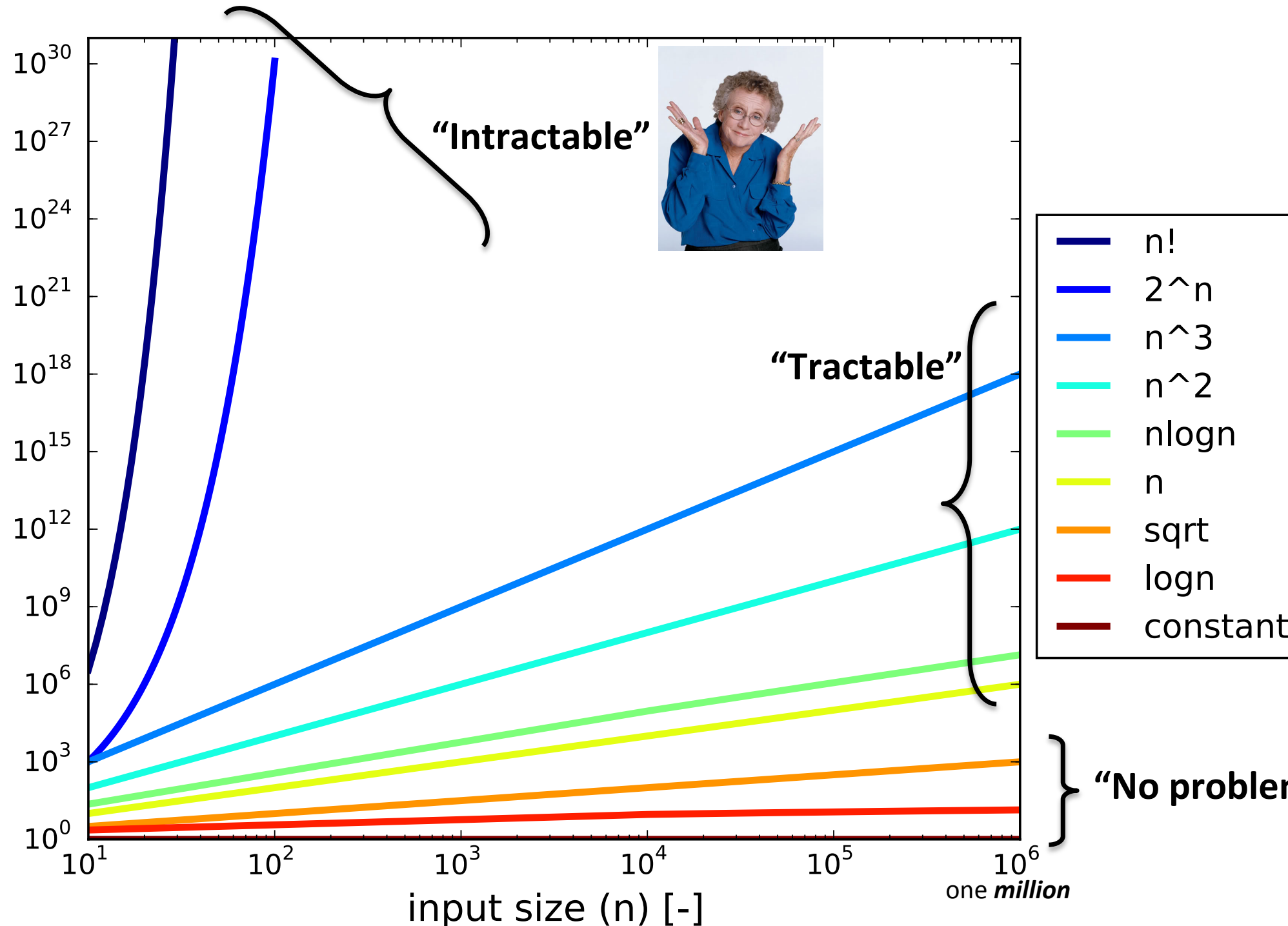
POWERED BY THE WOLFRAM LANGUAGE

Take Wolfram|Alpha anywhere.

PRO

Big O helps us talk about problem classes

Log scale, expanded range



How do we learn a new data structure?

What kinds of things do we want to know?

Order matters: Sequence of genetic mutations determines how cancer behaves

Date: February 11, 2015

Source: University of Cambridge

Summary: The order in which genetic mutations are acquired determines how an individual cancer behaves, according to new research.



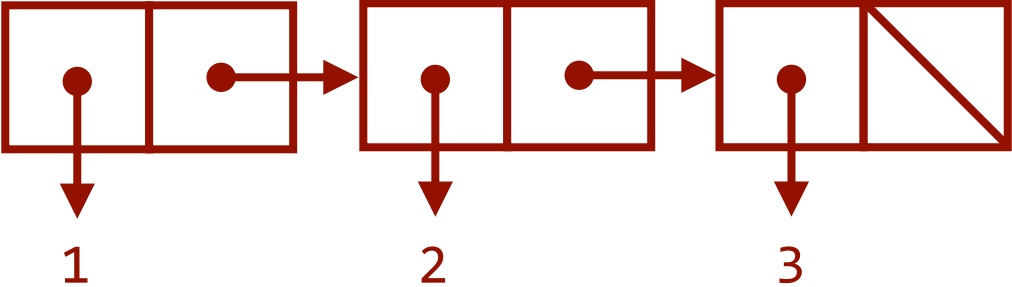
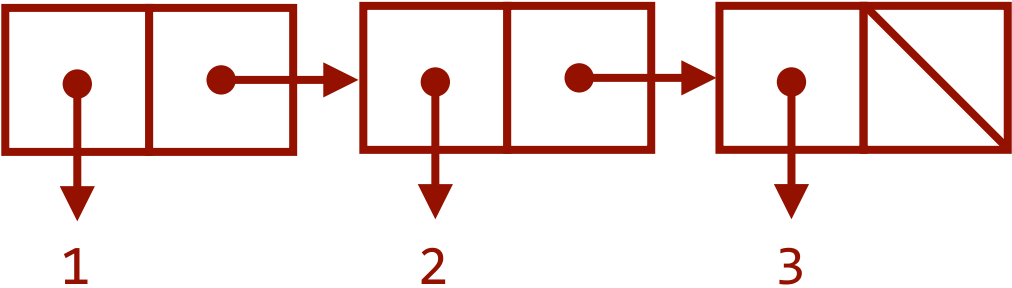
LUNCH	
Soup	☐ Creamy Chicken Soup
	☐ Roasted Garden Vegetable Soup  
	☐ Curried Carrot Soup  
	☐ Breadsticks 
Creations	☐ Made to Order Salad
Exhibition	☐ Beef Pho Noodle Bowl
Oven	☐ The Hawaiian
	☐ Housemade Pepperoni Pizza
	☐ Cheese Pizza 
	☐ Wild Mushroom & Pesto Flatbread Pizza
	☐ Hot Italian Sausage Pizza
Grill	☐ Burger Bar

A Step-By-Step Guide To Making A BuzzFeed Post

So you've [joined](#) the BuzzFeed Community and you want to make your own post, but don't know where to begin. Don't worry, we've got you covered!

posted on Oct. 14, 2014, at 8:37 a.m.

Creating lists in Racket

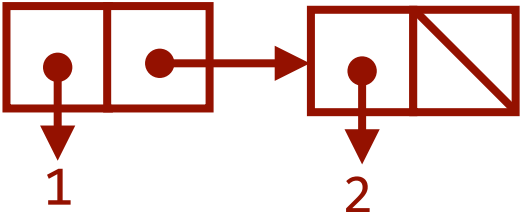
syntax what we <i>write</i>	printed representation what Racket <i>prints</i>	semantics what it <i>means</i>
empty <i>make an empty list</i>	> empty '()	
(list <value1> ... <valueN>) <i>make a list with N values</i>	> (list 1 2 3) '(1 2 3)	
(cons <value> <list>) <i>add an element to the front of a list</i>	> (cons 1 (list 2 3)) '(1 2 3)	

Creating lists: let's practice

write down the answers as either a drawing or a Racket expression

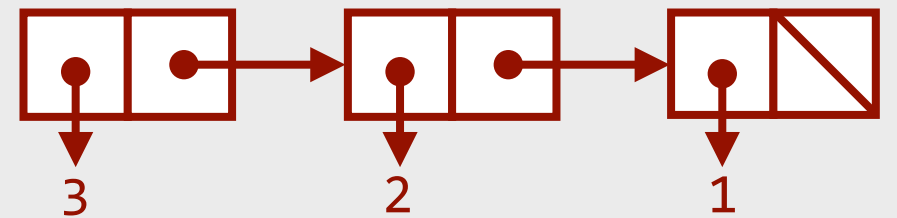
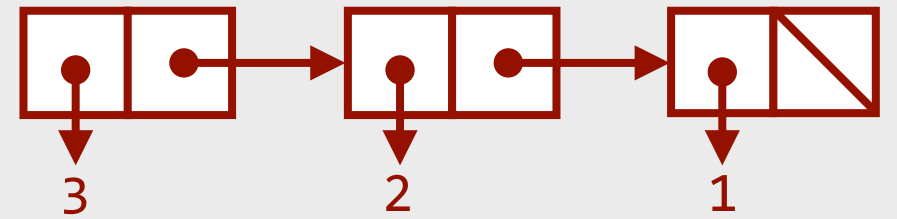
1. `(list 3 2 1)` ← draw the picture

2. `(cons 3 (list 2 1))` ← draw the picture

3.  ← write the expression

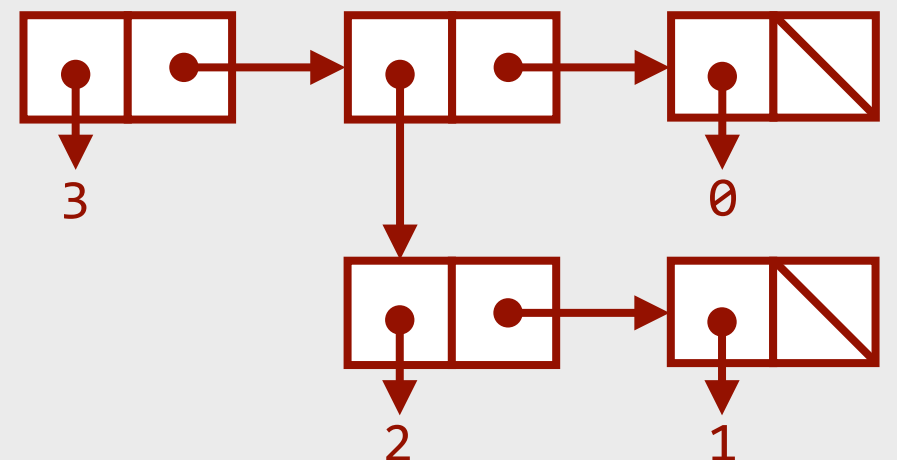
4. `'(1)` ← write the expression that makes Racket display this

5. `(list 3 (list 2 1) 0)` ← draw the picture



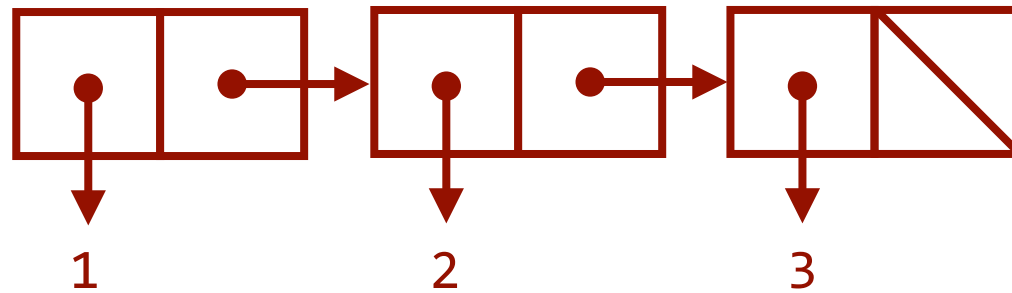
`(list 1 2)`

`(list 1)`



Aside: we don't actually *need* list!

list is “syntactic sugar” for one or more calls to cons



`(list 1 2 3)`

is the same as

`(cons 1 (cons 2 (cons 3 empty)))`

More syntactic sugar for lists

You can use Racket's display notation to construct a value

'()

'(1 2 3)

'(3 (1 2) 0)

is the same as

is the same as

is the same as

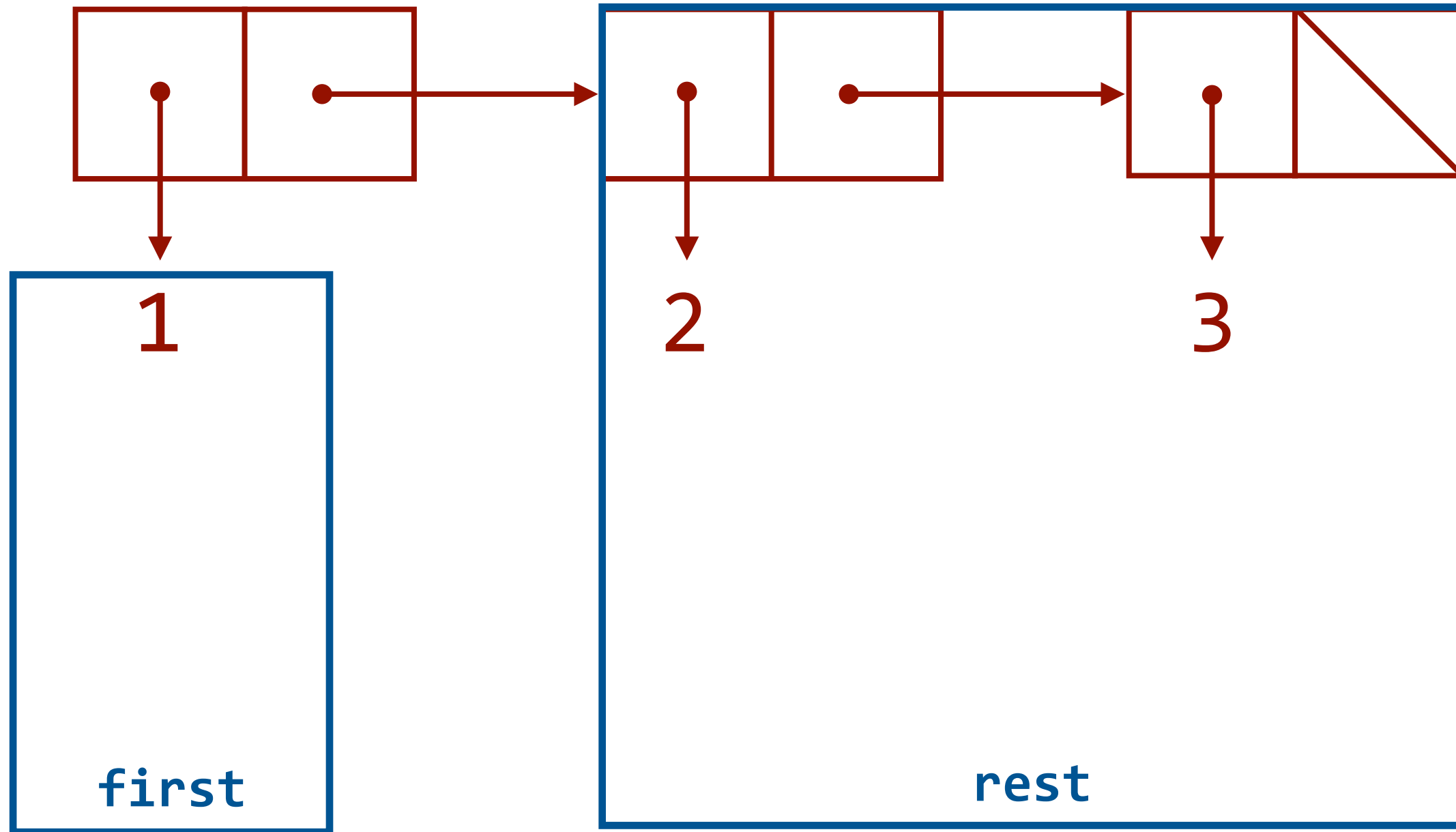
empty

(list 1 2 3)

(list 3 (list 1 2) 0)

But: be careful!
(more in a bit)

Accessing Racket lists



Accessing lists: let's practice

Assume the variable L has the value ' (1 2 3)'. Fill in the table.

result	expression that uses L to compute result
1	(first L)
' (2 3)	
2	
' (3)	

Accessing lists: let's practice

Assume the variable L has the value ' (1 2 3)'. Fill in the table.

result	expression that uses L to compute result
1	(first L)
' (2 3)	(rest L)
2	(first (rest L))
' (3)	(rest (rest L))

Watch out!

Don't do these things (and if you accidentally do, know how to recognize them)

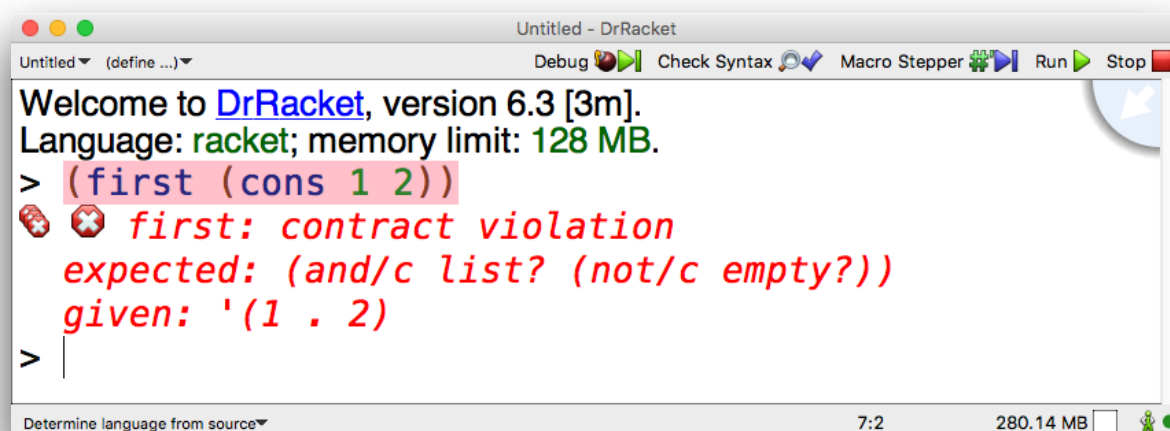
```
> (cons 1 2)  
'(1 . 2) ← not a list!
```

This expression builds a *pair*.

A pair is **not** a list.

You can't call `first` on it.

You can't call `rest` on it.



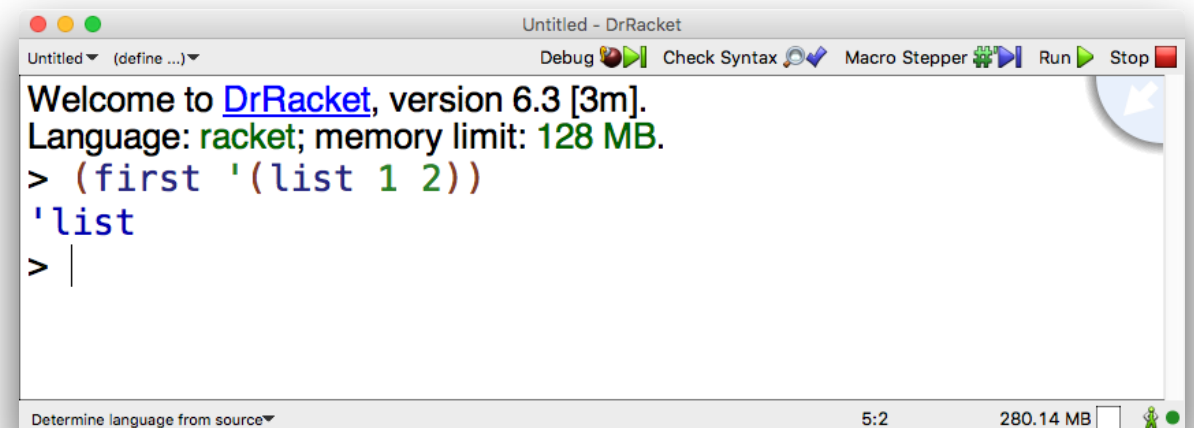
Untitled - DrRacket

```
Welcome to DrRacket, version 6.3 [3m].  
Language: racket; memory limit: 128 MB.  
> (first (cons 1 2))  
first: contract violation  
expected: (and/c list? (not/c empty?))  
given: '(1 . 2)  
>
```

Determine language from source 7:2 280.14 MB

```
> '(list 1 2)  
'(list 1 2)
```

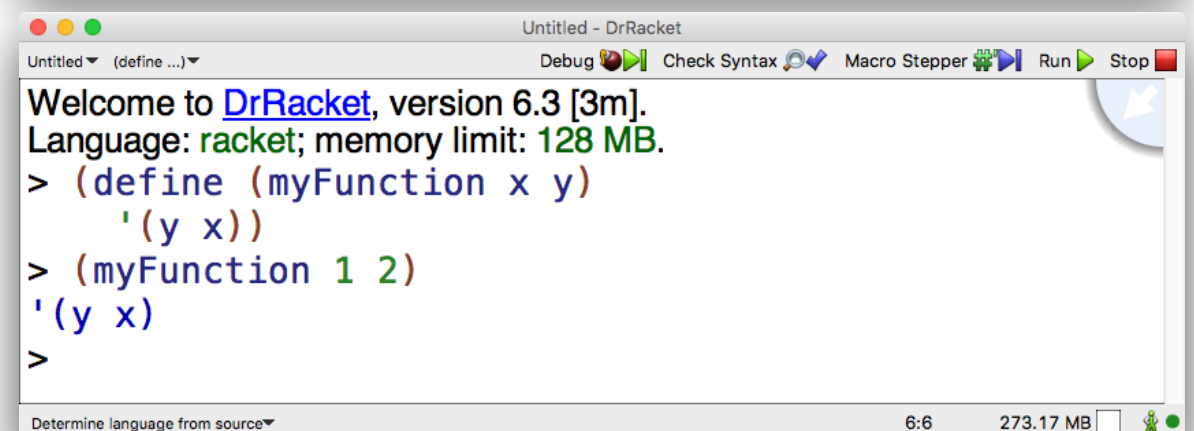
This expression builds a list
whose first element is 'list!



Untitled - DrRacket

```
Welcome to DrRacket, version 6.3 [3m].  
Language: racket; memory limit: 128 MB.  
> (first '(list 1 2))  
'list  
>
```

Determine language from source 5:2 280.14 MB



Untitled - DrRacket

```
Welcome to DrRacket, version 6.3 [3m].  
Language: racket; memory limit: 128 MB.  
> (define (myFunction x y)  
  '(y x))  
> (myFunction 1 2)  
'(y x)  
>
```

Determine language from source 6:6 273.17 MB

For more info, see: [docs.racket-lang.org/guide/Pairs Lists and Racket Syntax.html](https://docs.racket-lang.org/guide/Pairs_Lists_and_Racket_Syntax.html)



Racket lists: the basics

We'll use these built-in operations the *most*, when working with Racket lists

Constructors

to make an empty list: **empty**

to make a list with n elements: **list**

to add an element to the beginning of a list: **cons**

Selectors

to test if a list is empty: **empty?**

to get the first element of a list: **first**

to get a sublist, starting with the second element: **rest**

Inductive Data Types & Recursive Operations

(Putting Together & Taking Apart)

empty

cons

empty?

first

rest

Recursive functions over lists

```
;; Len
```

```
;; inputs: a list, L
```

```
;; outputs: the number of elements in the list
```

```
(define (len L)
```

)

Recursive functions over lists

;; Len

```
;; inputs: a list, L
```

```
;; outputs: the number of elements in the list
```

```
(define (len L)
```

)

```
; tests
```

```
(check-equal? (len '()) 0)
```

```
(check-equal? (len '(1 2 3)) 3)
```

```
(check-equal? (len '(1 2 3)) 1)
```

Recursive functions over lists

```
;; len  
;; inputs: a list, L  
;; outputs: the number of elements in the list  
(define (len L)
```

base case

)

recursive step

```
; tests  
(check-equal? (len '()) 0)  
(check-equal? (len '(1 2 3)) 3)  
(check-equal? (len '((1 2 3))) 1)
```

Recursive functions over lists

```
;; len  
;; inputs: a list, L  
;; outputs: the number of elements in the list  
(define (len L)  
  (if (empty? L)                                base case  
                                          ))          recursive step
```

```
; tests  
(check-equal? (len '()) 0)  
(check-equal? (len '(1 2 3)) 3)  
(check-equal? (len '((1 2 3))) 1)
```

Recursive functions over lists

```
;; len  
;; inputs: a list, L  
;; outputs: the number of elements in the list  
(define (len L)  
  (if (empty? L)                                base case  
    0  
    ))                                           recursive step
```

```
; tests  
(check-equal? (len '()) 0)  
(check-equal? (len '(1 2 3)) 3)  
(check-equal? (len '((1 2 3))) 1)
```

Recursive functions over lists

```
;; len  
;; inputs: a list, L  
;; outputs: the number of elements in the list  
(define (len L)  
  (if (empty? L)                                base case  
    0  
    (+ 1 (len (rest L))))                       recursive step  
  
; tests  
(check-equal? (len '()) 0)  
(check-equal? (len '(1 2 3)) 3)  
(check-equal? (len '((1 2 3))) 1)
```

Let's practice: sum



```
;; sum  
;; inputs: a list of integers, L  
;; outputs: the sum of the integers in L  
(define (sum L)
```

```
; tests  
(check-equal? (sum '()) 0)  
(check-equal? (sum '(1)) 1)  
(check-equal? (sum '(1 2 1)) 4)  
(check-equal? (sum '(1 2 3 4)) 10)
```

Let's practice: sum



```
;; sum  
;; inputs: a list of integers, L  
;; outputs: the sum of the integers in L  
(define (sum L)  
  (if (empty? L)  
    0  
    (+ (first L) (sum (rest L))))  
  
; tests  
(check-equal? (sum '()) 0)  
(check-equal? (sum '(1)) 1)  
(check-equal? (sum '(1 2 1)) 4)  
(check-equal? (sum '(1 2 3 4)) 10)
```

Let's practice: analyzing sum

what happens as the size of L grows very large?

```
(define (sum L)
  (if (empty? L)
      0
      (+ (first L) (sum (rest L)))))
```

- What mathematical tool should we use?
- What's the first question we should ask ourselves?
- What do you think is the answer?

Let's practice: analyzing sum

what happens as the size of L grows very large?

```
(define (sum L)
```

```
  (if (empty? L)
```

```
      0
```

```
      (+ (first L) (sum (rest L)))))
```

base case

recursive step

$$T(0) = 1 \text{ step}$$

$$T(n) = 1 + T(n-1) \text{ steps}$$

Recurrence equation solution:

$$t(n) = n + 1$$

Let's practice: remove

```
;; remove  
;;   inputs: an integer i  
;;           a list of integers, L  
;;   outputs: a new list with 1st occurrence  
;;            of i removed  
(define (remove e L)
```

```
; tests  
(check-equal? (remove 1 '()) '())  
(check-equal? (remove 1 '(1)) '())  
(check-equal? (remove 1 '(1 1)) '(1))  
(check-equal? (remove 4 '(1 2 3)) '(1 2 3))
```

Let's practice: remove

;; remove

;; inputs: an integer i

;; a list of integers, L

*;; outputs: a new list with 1st occurrence
;; of i removed*

(define (remove e L)

(cond [(empty? L) '()]

[(= (first L) e) (rest L)]

[else (cons (first L) (remove e (rest L)))]))

; tests

(check-equal? (remove 1 '()) '())

(check-equal? (remove 1 '(1)) '())

(check-equal? (remove 1 '(1 1)) '(1))

(check-equal? (remove 4 '(1 2 3)) '(1 2 3))